# LCS: a Linux-based Content Switch

C. Edward Chow, Weihong Wang, and Chandra Prakash
Department of Computer Science
Univ. of Colorado at Colorado Springs
Colorado Springs, CO 80933-7150, USA
Tel: (719) 262-3110
Fax: (719) 262-3369
Email: {chow, wwang, cprakash}@cs.uccs.edu

**Abstract**

In this paper we present the design of a Linux-based content switch, propose a pre-allocate server scheme improving the TCP delay binding, discuss the lessons learnt from the implementation of the content switch, and suggest system components/modules for high speed content switch processing. A content switch routes packets based on their headers in the upper layer protocols and the payload content. We discuss the processing overhead and the content switch rule design. Our content switch can be configured as the front end dispatcher of web server cluster and as a firewall. By implementing the http header extraction and xml tag extraction, the content switch can load balancing the requests based on the file extension or the text pattern in the url, and routes big purchase requests in XML to faster servers in e-commerce systems. The rules and their content switch rule matching algorithm are implemented as a module and hence can be replaced without restarting the system. With additional SMTP header extraction, it can be configured as a spam mail filter or virus detection/removal system.

**Keywords:** Internet Computing, Cluster Computing, Content Switch, Network Routing

## 1 Introduction

With the rapid increase of Internet traffic, the workload on servers is increasing dramatically. Nowadays, servers are easily overloaded, especially for a popular web server. One solution to overcome the overloading problem of the server is to build scalable servers on a cluster of servers [1, 2]. A load balancer is used to distribute incoming requests among servers in the cluster. Load balancing can be done in different network layers. A web switch is an application level (layer 7) switch, which examine the headers from layer 2 all the way to the HTTP header of the incoming request to make the routing decisions. By examining the HTTP header and its content, a web switch can provide a higher level of control over the incoming web traffic, and make decision on how individual web pages, images, and media files get served from the web site. This level of load balancing can be very helpful if the web servers are optimized for specific functions, such as image serving, SSL (Secure Socket Layer) sessions or database transactions.

By having a generic header/content extraction module and rule matching algorithm, a web switch can be extended as a content switch [4,5] for route packets including other application layer

protocols, such as SMTP, IMAP, POP, and RTSP. By specifying different sets of rules, the content switch can be easily configured as a load balancer, firewall, policy-based network switch, a spam mail filter, or a virus detection/removal system.

Traditional load balancers known as Layer 4 (L4) switches examine IP, TCP, and UDP headers, such as IP addresses or TCP and UDP port numbers, to determine how to route packets. Since L4 switches are content blind, they cannot take the advantages of the content information in the request messages to distribute the load. For example, many e-commerce sites use secure connections for transporting private information about clients. Using SSL session IDs to maintain server persistence is the most accurate way to bind all a client's connections during an SSL session to the same server. A content switch can examine the SSL session ID of the incoming packets, if it belongs to an existing SSL session, the connection will be assigned to the same server that was involved in previous portions of the SSL session. If the connection is new, the web switch assigns it to a real server based on the configured load balancing algorithm, such as weighted least connections and round robin. Because L4 switches do not examine SSL session ID, which is in layer 5, so that they cannot get enough information of the web request to achieve persistent connections successfully. Web switches can also achieve URL-based load balancing. URL based load-balancing looks into incoming HTTP requests and, based on the URL information, forwards the request to the appropriate server based on predefined polices and dynamic load on the server.

XML are proposed to be the language for describing the e-commerce requests. A web system for e-commerce system should be able to route requests based on the value in the specific tag of a XML document. It allows the requests from a specific customer, or of different purchase amount to be processed differently. The capability to provide differential services is the major feature provided by the web switch. Intel XML distributor is such an example, it is capable of routing the request based on the url and the XML tag sequence [14].

The content switching system can achieve better performance through load balancing the requests over a set of specialized web servers, or achieve consistent user-perceived response time through persistent connections, also called sticky connections.

### *1.1 Related Content switching Techniques*

Application level proxies [6,7] are in many ways functionally equivalent to content switches. They classify the incoming requests and match them to different predefined classes, then make the decision whether to forward it to the original server, or get the web page directly from the proxy server based on proxy server's predefined behavior policies. If the data are not cached, the proxy servers establish two TCP connections –one to the source and a separate connection to the destination. The proxy server works as a bridge between the source and destination, copying data between the two connections. Our proposed Linux-based Content Switch (LCS) is implemented in kernel IP layer. It reduces the protocol processing time and provides more flexible content switching rules and integration with load balancing algorithms.

Microsoft Windows2000 Network Load Balancing (NLB) [3] distributes incoming IP traffic to multiple copies of a TCP/IP service, such as a Web server, each running on a host within the cluster. NLB transparently partitions the client requests among the hosts and lets the clients access the cluster

using one or more "virtual" IP addresses. With NLB, the cluster hosts concurrently respond to different client requests, even multiple requests from the same client.

Linux Virtual Server(LVS) is a load balancing server which is built into Linux kernel [2]. In the LVS server cluster, the front-end of the real servers is a load balancer, also called virtual server, that schedules incoming requests to different real servers and make parallel services of the cluster to appear as a virtual service on a single IP address. A real server can be added or removed transparently in the cluster. The load balancer can also detect the failures of real servers and redirect the request to an active real server. Our LCS content switch is based on Linux LVS code and when no content switch rules match with the incoming packet, the packet is routed based on the layer 4 LVS scheduling policy.

The rest of the paper is organized as follows: In Section 2, we present the basic architecture and modules of the Linux Content Switch. Section 3 presents the pre-allocate server scheme for improving the TCP delayed binding. Section 4 discusses the problems encountered in the design of LCS and their solutions. System components and modules for high speed content switch processing are suggested. Section 5 shows the content switch rule design. The performance results of our LCS implementation are presented in Section 6. Section 7 is the conclusion.

## 2 Linux-based content switch design

The Linux-based Content Switch (LCS) is based on the Linux 2.2-16 kernel and the related LVS package. LVS is a Layer 4 load balancer which forwards the incoming request to the real server by examining the IP address and port number using some existing schedule algorithm. LVS source code is modified and extended with new content switching functions. LCS examines the content of the request, e.g., URL in HTTP header and XML payload, besides its IP address and port number, and forwards the request to the real servers based on the predefined content switching rules. Content switch rules are expressed in term of a set of simple if statements. These if statements include conditions expressed in terms of the fields in the protocol header or pattern in the payload and branch statements describing the routing decisions. Detailed of the content switching rules are presented in Section 5.

Figure 1 shows the main architecture of LCS. Here *the Content Switch Schedule Control* module is the main process of the content switch and is used to manage the packet follow. *Routing Decision, INPUT rules, FORWARD rules and OUTPUT rules* are all original modules in Linux kernel. They are modified to work with the Content Switch Schedule Control module. The *Content switch Rules* module is the predefined rule table. The *Content switch schedule control* module will use this information to control the flow of the packets. The *Connection Hash Table* is used to speed up the forwarding process by retrieving the real server assignment based on the packet header information. The *LVS Configuration* and *Content Switch Configuration* are user space tools used to define the content switch server clusters and the content switch rules.

LCS uses Linux Network Address Translation (NAT) approach for routing packets between the client and the real server. When an incoming packet arrives at IP layer, *Routing Decision* function is called to check if the packet is destined to local or remote host. If the packet is for the local host, *INPUT RULES* function is called to deliver the packet to the *Content Switch Schedule Control* module.

Otherwise *FORWARD RULES* function is called to pass the packet to the *Content Switch Schedule Control* module. The *Content Switch Schedule Control* module will check the *Connection Hash Table* to see if it belongs to an existing connection. If the packet is a new request, the *Content Switch Schedule Control* module will extract the header/content of the request and apply *Content Switch Rules* on it to choose a real server for this request. Also a new hash table for this connection is created, and then *FORWARD RULES* function is called to forward this request to the chosen server. If the packet is from an existing connection, The *Content Switch Schedule Control* module will call *FORWARD RULES* function to forward the packet based on the information in hash table.

Figure 1. LCS Architecture

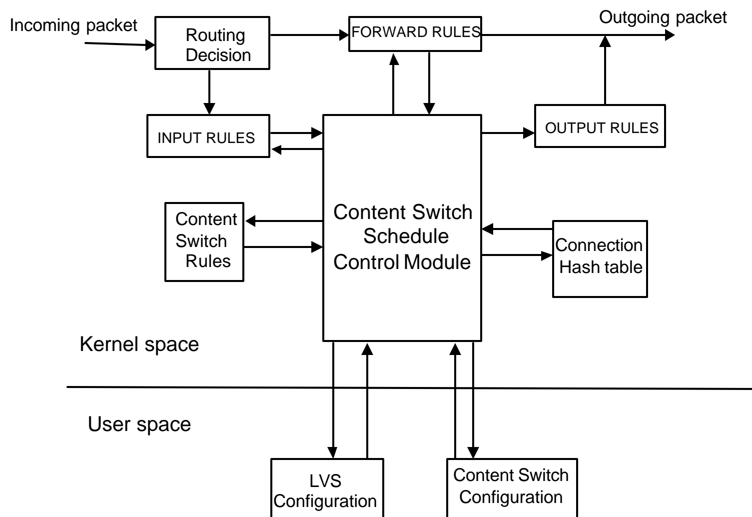


Figure 2 shows the input output processing of the content switch in IP layer of Linux Network Software. **cs_infromclient** manages the packet from the client to the content switch; **cs_infromserver** handles the packet from the server back to the client.

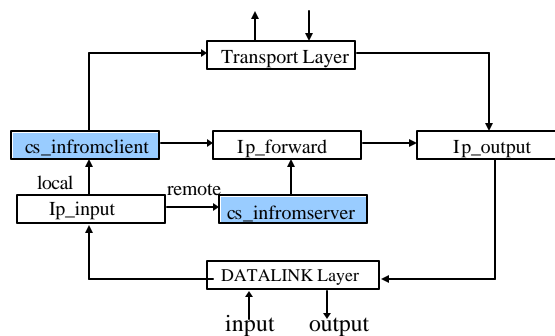


Figure 2. Content switch functions added to IP layer in Linux network software.

## 3 Improve TCP Delayed Binding with Pre-allocate Server Scheme

Many upper layer protocols utilize TCP protocol for reliable orderly message delivery. The TCP connection will be established via a three way handshake, and the client will not deliver the upper layer information until the three way handshake is complete. The content switch then selects the real

server, establishes the three way handshake with the real server, and serve a bridge that relays packets between the two TCP connections. This is called *TCP delayed binding*.

### 3.1 The message exchange sequence in TCP Delayed Binding

Because the client established the connection with the content switch, it accepts the sequence number chosen by the content switch and when the packets come from real server to client, content switch must change their sequence numbers to the ones that client expects. Similarly, the packets from client to server are also changed by content switch. By doing the packet rewriting, the content switch "fools" both the client and real server, and they communicate with each other without knowing the content switch is playing the middleman. Detailed sequence number rewriting process is shown below in Figure3.

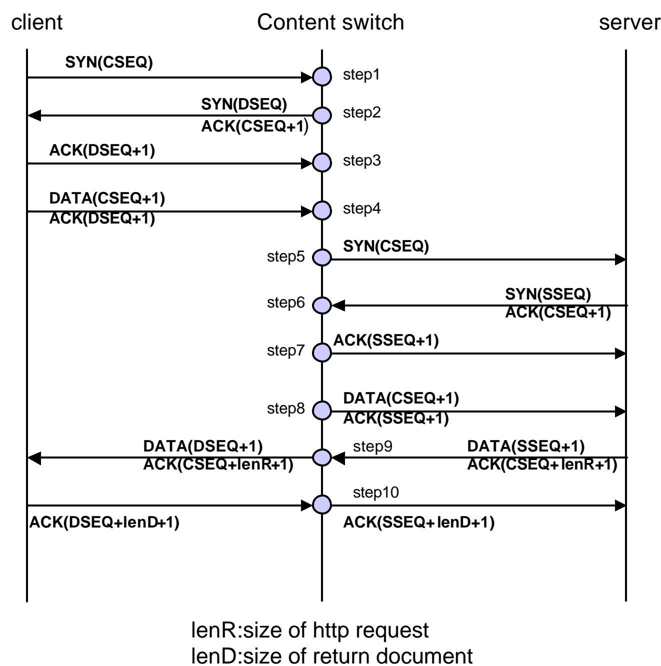


Figure 3. Message Exchange Sequence in TCP Delay Binding.

**Step1-Step3**: The process is the standard TCP three way handshake between the client and the content switch. The client and content switch commit their initial sequence numbers as CSEQ and DSEQ.

**Step4:** The client sends the application level request data to the content switch. The content switch chooses a real server based on the request data to server this request. The request data may contain more than one IP packet, so the content switch need to get all the IP packets before invoking rule matching algorithm.

**Step5-Step7:** The content switch establishes a TCP connection with the real server. The content switch forwards the SYN request from the client to the server using its original initial sequence number CSEQ, the server commits its own initial sequence number SSEQ.

**Step8**: The DATA message with the request is forwarded from the content switch to the server. The original sequence number is kept while the ACK sequence number is changed from acknowledge number of the content switch (DSEQ+1) to that of the server (SSEQ+1).

**Step9**: For the return data from the server to the client, the sequence number needs to be changed to that associated with the content switch. For a large document, several packets are needed. Push flags in the TCP header are typically set on the follow up packets, so the client TCP process will deliver them immediately to the upper layer process.

**Step10**: For the ACK packet from the client to the content switch, the ACK sequence number is changed from the one acknowledging the content switch to that acknowledging the server.

Delayed binding is the major technique used in the content switch design. To maintain correct connection between the client and the server, the content switch must adjust the sequence number in every packet for each direction. This requests that all the subsequent packets go through the content switch and have their sequence numbers changed. As many other existing content switch products, the content switch design presented in this paper uses NAT (Network Address Translation) approach.

### *3.2 Design of Pre-allocate Server Scheme*

We implemented a heuristic solution to improve the TCP delayed binding problem, where the client and real server mapping is pre-allocated and stored in a hash table with (hash) key as the client address. When a client sends a request to the content switch for the first time, there will not be any entry in the hash table, and the request will go via the normal data buffering and rule matching processing. An entry will be added to the pre-allocate hash table with the client IP address as key and the real server address as data. When the same client sends the next request, an entry will be found in the pre-allocate hash table, and the client request will be directly forwarded to the corresponding real server. This reduces the rule matching overhead.

Figure 4a shows the modified delay binding in the pre-allocate scheme when the pre-allocate server is the right one. Figure 4b shows the message exchange among the pre-allocate server, the right server, the content switch, and the client, when the guess it wrong. Note that when the guess it right, the web access can be complete in six steps instead of ten steps, and there is no need for the sequence number modification for Step5 and Step6. When the direct routing or IP tunnel scheme is used instead of NAT, the return document can be sent directly to the client and thus reduce the processing overhead at the content switch.

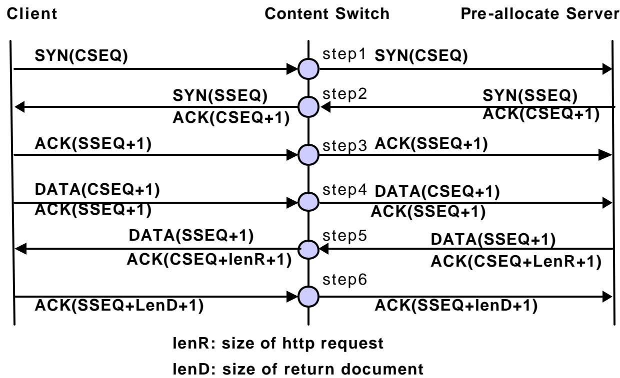


Figure 4a. Pre-allocate server scheme when guess it right.

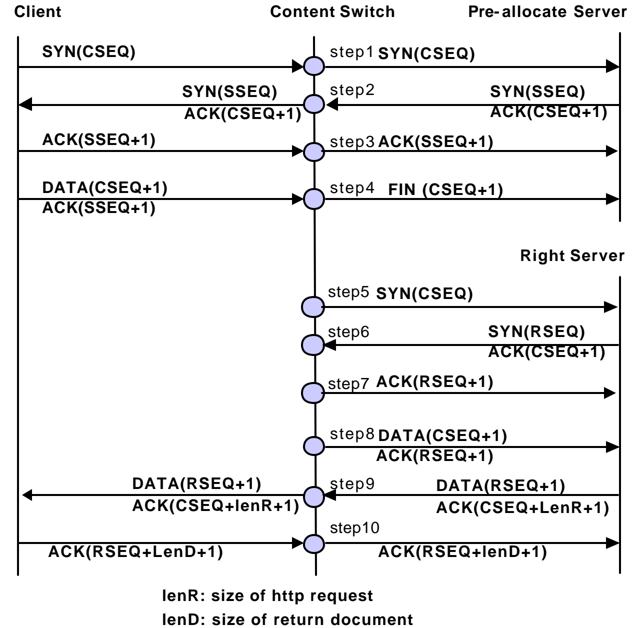


Figure 4b. Pre-allocate server scheme when guess it wrong.

For subsequent requests, when there is a matching hash table entry found in the pre-allocate hash table, it may happen that the real server specified by the matching entry may not be the correct real server for that request. In that case the pre-allocate scheme degenerates to the default TCP delayed binding, where rule matching is done for the client request. The worst case scenario in the pre-allocate scheme is where the real server specified in the matching hash table entry happens to be wrong choice. Here it mandates that the client data are buffered as done in the default scheme.

The content switch must examine the response from the real server specified in the matching hash table entry, before applying the degenerate rule matching. If the response does not contain HTTP response code 200, then only content switch switches to the default scheme. If the response code is 200, we then free up the queued client request. This provide a retry mechanism for improving the probability of document delivery.

In our implementation if the real server specified in the matching hash table entry and the real server selected via rule matching after a wrong pre-allocate guess are same, we allow the response from the wrongly guessed real server to be forwarded to the client.

# 4 Problems and Solutions for Content Switch Design

In this section we discuss the content switch design issues related to content switch processing and client request buffering.

## *4.1 Handling request with multiple packets*

If the client's request is too big to fit in one TCP segment, the content switch has to wait for all the segments that comprise that request before commencing the rule matching. This is especially true

of non-idempotent HTTP requests like PUT and POST, and for e-commerce application with large XML request. This further gives rise to the following sub-problems that we had to account for:

**Determine the content length**

We had to determine the content length of the variable incoming data stream in order to flag end of client request. The content length information of such request can be obtained from the "Content-Length" fields in the HTTP header. However, the value of the content length itself can span across multiple segments as shown in the example below:

TCP Segment n contains:

```
POST /cgi-bin/cs622/purchase.pl HTTP/1.0\r\n

Referer: http://archie.uccs.edu/~acsd/lcs/xmldemo.html\r\n
Connection: Keep-Alive\r\n
User-Agent: Mozilla/4.75 [en] (X11; U; Linux 2.2.16-22enterprise i686) \r\n
Host: viva.uccs.edu\r\n
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*\r\n
Accept-Encoding: gzip\r\n
Accept-Language: en\r\n
Accept-Charset: iso-8859-1,*,utf-8\r\n
Content-type: application/x-www-form-urlencoded\r\n
Content-length: 7
```

TCP Segment n+1 contains:

```
53\r\n
data (753 bytes)
```

As seen in the above example, the individual bytes of the content length are split across two consecutive TCP segments, the first segment contains 7 and the next segment contains the remaining two byte, i.e., 53. This is true for any field within the HTTP request header, even for the sequence of data bytes that form the "Content-Length" string.

**Fragmentation of application level content**

After the content length is determined, the content switch can then wait for all the packets of the same request. Typically, these packets are saved in different memory area. In Linux, they are saved in skbuf structures linked by double link list. Each of these data structures contains the timestamp, IP/TCP headers, followed by the content payload. Therefore, the actual content is fragmented and spread out in the network buffer. Extracting URL field in the HTTP request is easy, since it is in the first packet. But for extracting other meta-headers and especially the XML tag values in the content field of the HTTP request, the fragmentation of the TCP payload content post difficult challenging problem for the content switch designer. One approach is to concatenate all individual non-contiguous TCP segments back to back into one coherent buffer, that can then be used for XML parsing, or pattern matching. Another approach is to redesign the XML parsing or pattern matching so

that they can work with data that spread across several segments. A specialized memory address mapping hardware similar to the translation look-aside cache used in virtual memory system can also help speed up this type of packet processing.

The first approach requires the expensive memory copying and uses additional memory. The original TCP segments are not released after the concatenation of their payload content, since once the real server is selected, these TCP segments will be modified and sent to the chosen real server. The modification includes the destination IP address field, possibly the TCP port field, the ACK sequence number, and very importantly the checksum.

While buffering client data, the content switch has to send ACK's for the segments that comprise the client request, otherwise the client TCP will assume the server is dead or is very slow, and will not send subsequent packets. This is achieved by invoking appropriate ACK sending routines from the IP layer of the content switch.

For large sized (> 40K) client requests, we also observed some of the relayed segments were dropped by the chosen real sever. Further analysis indicate that the problem is due to the segment relay by the content switch is implemented in IP, instead of TCP layer. The data sending was done continuously from the queued buffers without considering the window advertised by the TCP stack of the real server. This flooding of data caused the real server to drop some of the received TCP packets. It was observed that the acknowledgment number sent by real server was held constant, even though the content switch had emptied all buffered data. The result was that there was no response seen from real server, as if it had not acknowledged receipt of all data. This problem was solved by having the content switch keep track of the acknowledgment number. When the acknowledgment sent by real server was less than or equal to the sequence number of the last sent packet, the last sent packet was retransmitted. This retransmission helped alleviate packet flooding at the real server and ensure all client data are properly received.

### *4.2 Handle Different Data Encoded Methods*

There are two basic ways for submitting the XML-based request to the web server. One is to use the form with text input or text area input. The other is to submit it as XML document. When submitting it with the form, the XML request data are encoded using the x-www-form-urlencoding method and the “Content-Type” meta-header will have the value of “x-www-form-urlencoded”. When submitting it as XML document, the “Content-Type” meta-header will have the value of “text/xml” and the content is submitted with the plain text without further encoding. With the latter encoding type, all special characters like line feed (\n), carriage return (\r), left anchor (<) and right anchor (>) etc. retain their ASCII representation. In the former encoding type, the special characters have encodings like "%XX", where XX is the hexadecimal representation of ASCII value of that special character. For example, for the "x-www-form-urlencoded" encoding type, the values for the indicated special characters will be "%0A", "%0D", "%3C" and "%3E" respectively. Hence, the rule matching module should correctly parse the XML content of the client request depending on the content type.

### *4.3 Allow Referencing Specific XML Tags*

The rule specification scheme should be flexible enough to account for exact tag name or rule field indicated in the rule specification. Here is an example that illustrates this point. Consider the XML document:

```
<purchase>
  <customerName>CCL</customerName>
  <customerID>111222333</customerID>
  <item>
    <productID>309121544</productID>
    <unitPrice>5000</unitPrice>
    <subTotal>50000</subTotal>
  </item>
  <item>
    <productID>309121538</productID>
    <unitPrice>200</unitPrice>
    <subTotal>2000</subTotal>
  </item>
  <totalAmount>52000</totalAmount>
</purchase>
<purchase>
<customerName>CDL</customerName>
  <customerID>111222444</customerID>
  <item>
    <productID>30913555</productID>
    <unitPrice>3000</unitPrice>
    <subTotal>20000</subTotal>
  </item>
<totalAmount>20000</totalAmount>
 </purchase>
```

In the above XML document, some of the tags are repeated, e.g., purchase, item, totalAmount. Hence a rule syntax is needed to allow for selecting a particular set of tags in the rule set. Here is an example of a scheme that addresses this problem. To specify a rule based on subTotal value present in the second item tag within the first purchase tag, the condition of the rule will be specified as 'purchase:1.item:2.subtotal > 5000". As another example, 'purchase:2.totalAmount < 15000' specifies the condition of a rule based on the totalAmount tag present within the second purchase tag.

### *4.4 Handle Long Transactions in SSL and Email network services*

In our Linux-based Content Switch, the content/header extraction and rule matching are performed in the kernel to avoid unnecessary copying. However, we have found that for network services that require long computation and interface with other packages, some of the packet processing functions are better handled at the application level. For example, there are a lot of packages, including McAfee's uvscan and AMAVis scanmail, mutt (recombine email component), for detecting and removing email virus, but almost all of them are implemented in application level and interact with the sendmail program. It will require significant effort to rewrite them as kernel modules. Same observations were derived on SSL processing.

SMTP goes through long message exchange between the client and the server, where the client sends a sequence of messages including HELO, MAIL FROM, RCPT TO, Data, followed by the actual body of the message. The server will respond with specific code and confirmation message.

Therefore the important email addresses for the sender and the receiver will appear at different stages of the transaction. The content switch needs to be able to store these messages in the buffer. Once the related header information is extracted and rules matched, these messages will be forwarded to the real mail server. For spam mail removal, the sending email address is extracted from the MAIL FROM message. For incoming email load balancing, the receiving email address is extracted from the RCPT TO message. Compared with SMTP, the processing of IMAP or POP is much simpler, since we only need to wait for the login in USER message for load balancing rule matching, but they have the same requirement for storing and forwarding the message sequence to the real server.

### *4.5 Handle Multiple Requests in a Keep-Alive Session*

Most browsers and web servers support the keep-alive TCP connection. It allows a web browser to request documents referred by the embedded references or hyper links of the original web page through this existing keep alive connection without going through long three way handshake. It is a concern that different requests from the same TCP connection are routed to different web servers based on their content. The challenge here is how the content switch merges the multiple responses from different web servers to the client transparently using the keep alive TCP connection. Figure 5 shows the situation where different requests from one TCP connection go to different web servers through the content switch.

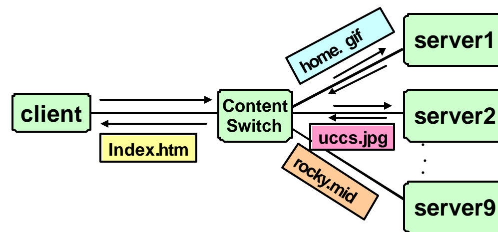


**Figure 5. Multiplexing Return Document into a Keep-Alive Connection**

If the client sends http requests within one TCP connection to the content switch, then the content switch can route these requests to three different web servers based on their contents and it is possible the return documents of those request will arrive at the content switch out of order. The content switch must be able to handle this situation.

The brute force solution will be to discard the early requests. One possible solution is to buffer the responses of the later request at the content switch so that they return in the same order as their corresponding requests. The drawback is that it significant increases the memory requirement of the content switch. The other solution is to calculate the size of the return documents and adjust the sequence number accordingly. It avoids the buffer requirement and the later requests will be sent with the starting sequence number that leaves space for those slow return documents. The drawback here is that the content switch needs to have the directory information of the server and how they map the request into the actual path of the file system.

# 5 The Content Switching Rule Design

## *5.1 LCS Content Switch Rule*

LCS rules are defined using C functions. The syntax of the rules is as follow:

RuleLabel: if (condition) { action1} [else { action2}].

Examples:

```
R1: if (xml.purchase/totalAmount > 52000) { routeTo(server1, STICKY_IP_PORT); }
R2: if (strcmp(xml.purchase/customerName, "CCL") = = 0) {
        routeTo(server2, NONSTICKY); }
R3: if (strcmp(url, "gif$") = = 0) { routeTo(server3, NONSTICKY); }

R4: if (srcip = = "128.198.60.1" && dstip = = "128.198.192.192" &&
        dstport = = 80) { routeTo(LBServerGroup, STICKY_IP); }

R5: if (match(url, "xmlprocess.pl")) { goto R6; }

R6: if(xml.purchase/totalAmount > 5000){routeTo(hsServers, NONSTICKY);}
        else {routTo(defaultServers, NONSTICKY); }
```

The rule label allows the use of goto and makes referencing easier. We have implemented match() function for regular expression matching and xmlContentExtract() for XML tag sequence extraction in the content switching rule module. The rule is designed as a dynamic kernel module. So it can be edited at running time without recompiling the kernel. To update to a new rule set, "rmmod" is called to removed the current rule set, and the content switch schedule control module will call a default function NO_CS() to schedule the requests using round robin algorithm, weighted connection, or weighted least connection. The content switch uses "insmod" command to insert the new rule module.

## *5.2 Support for Sticky Connections*

In LCS, there are three different options related to the sticky connections. These options are *STICKY_IP_PORT, STICKY_IP* and *NONSTICKY.* With the option *STICKY_IP_PORT,* if the condition is true, all the following packets with the same IP addresses and TCP port numbers will be routed to the same server directly without carrying out the rule matching process. This option will route all the requests in one TCP keep-alive connection to the same server. And the option *STICKY_IP* will stick all the packets with the same IP addresses to the same server. This option will route all the request from the same client to the same server. The option *NONSTICKY,* specifies the connection to be a non-sticky connection, so either the request from the same connection, or the new connection all need to go through the rule matching for selecting the real server.

## *4.3 Content Switch Rule Matching Algorithm*

Rule matching algorithm directly affects the performance of the content switch. It is related to the packet classification techniques [11,12.13]. In layer 4 switching, the switch only examines the IP

address and port number of the packet, which are in the fixed fields. So the rule matching process can be speed up by easily using the hash function. In content switch, the higher layer content information is needed. These information such as URL, HTTP header or XML tag are not from the fixed fields and have varying length. It is hard to build a hash data structure to speed the searching process. In our prototype, we have observed significant packet processing time. It is therefore crucial to improve the performance of the rule matching algorithm, to emphasize the differential treatment of packets and the flexibility to add other functions such as spam mail filtering. The order of the rules also affects the content switch performance. One way to improve the rule matching, is to set flags based on the packet type and organize the rule set by checking these flag first and skip rules that do not match the packet type. Detecting Conflicts among rules in the rule set is a challenging and important research issue.

# 6 LCS Performance Result

Since the content switch examines the content of the request before it forwards the request data to the real server, it introduces more overhead than the layer 4 load balancing method. For all the tests we have performed, we would like to find out what parameters affect the performance of the content switch. A LCS testbed was set up where a HP Vectra workstation with 240 MHz Pentium Pro Processor and 128 MB memory is used as the content switch, and four real servers are connected with the content switch in the same subnet.

Figure 6a shows the measured response time for the rule matching process inside the content switch kernel when a different number of rules are used. As shown in Figure 6a, the more rules the content switch has, the longer the process time. This is because the larger number of rules, it will take longer time to perform the rule matching. A more efficient algorithm will help to improve the rule matching performance.

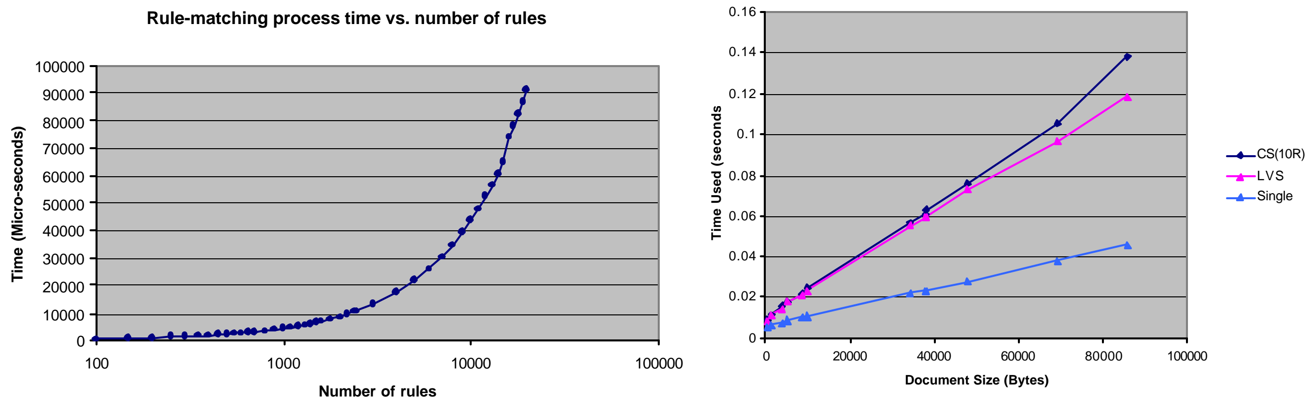


**Figure 6a. Rule process time vs. number of rules.** **Figure 6b. Rule process time vs. XML file size**

Figure 6b compares the impact of XML document processing overheads on LCS, LVS, and a single web server. Here we are interested in finding the impact of TCP delay binding overhead on LCS, and therefore we did not configure LCS to match any rule. It indicates the TCP delay binding incurred very small amount of additional processing compared to the overall processing time. The difference between LVS and that of a single server is the redirect routing processing time.

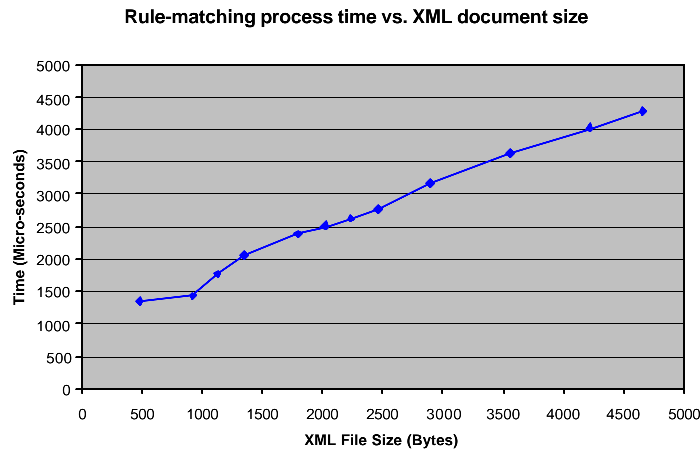


Figure 6c. Rule process time vs. XML file size.

Figure 6c shows the impact of XML document size on LCS processing time. If the request contains an XML document and the rules contain conditions related to the XML tag value, the process time varies with the XML document size. If rules are defined to choose the real server based on XML tag values, the rule-matching process will need to parse an XML request to find the XML tag value. The XML parsing process uses recursive algorithm, so the process time increases dramatically with the size of the XML document.

To evaluate the performance of the pre-allocate server scheme, a testbed with one content switch and two real server was set up with the following configurations:

| Machine Spec | IP Address | OS | Web Server |
|---|---|---|---|
| viva.uccs.edu<br>P5 240MHz 128MB<br>(Content Switch) | 128.198.192.192 | Redhat 6.2 running<br>LCS0.2 kernel based on<br>Linux 2.2-16-3 | Apache 1.3.14 |
| ace.uccs.edu<br>P5 166MHz 64MB<br>(Real Server 1) | 128.198.192.198 | Redhat 6.2 running<br>LCS0.2 kernel based on<br>Linux 2.2-16-3 | Apache 1.3.14 |
| vinci.uccs.edu<br>P5 240 MHz 128MB<br>(Real Server 2) | 128.198.192.193 | Redhat 6.2 running<br>LCS0.2 kernel based on<br>Linux 2.2-16-3 | Apache 1.3.14 |

We compared the response times of various document size between basic TCP delayed binding scheme and the pre-allocate scheme with the following set of series as shown in Figure 7:

Series 1 - Basic scheme with no rule matching module inserted, i.e., using default IPVS.

Series 2 - Basic scheme with the rule matching module inserted.

Series 3 - Pre-allocate scheme with all hits, i.e., where all pre-allocate guesses were correct.

Series 4 - Pre-allocate scheme with all misses, i.e., where all pre-allocate guesses were wrong.

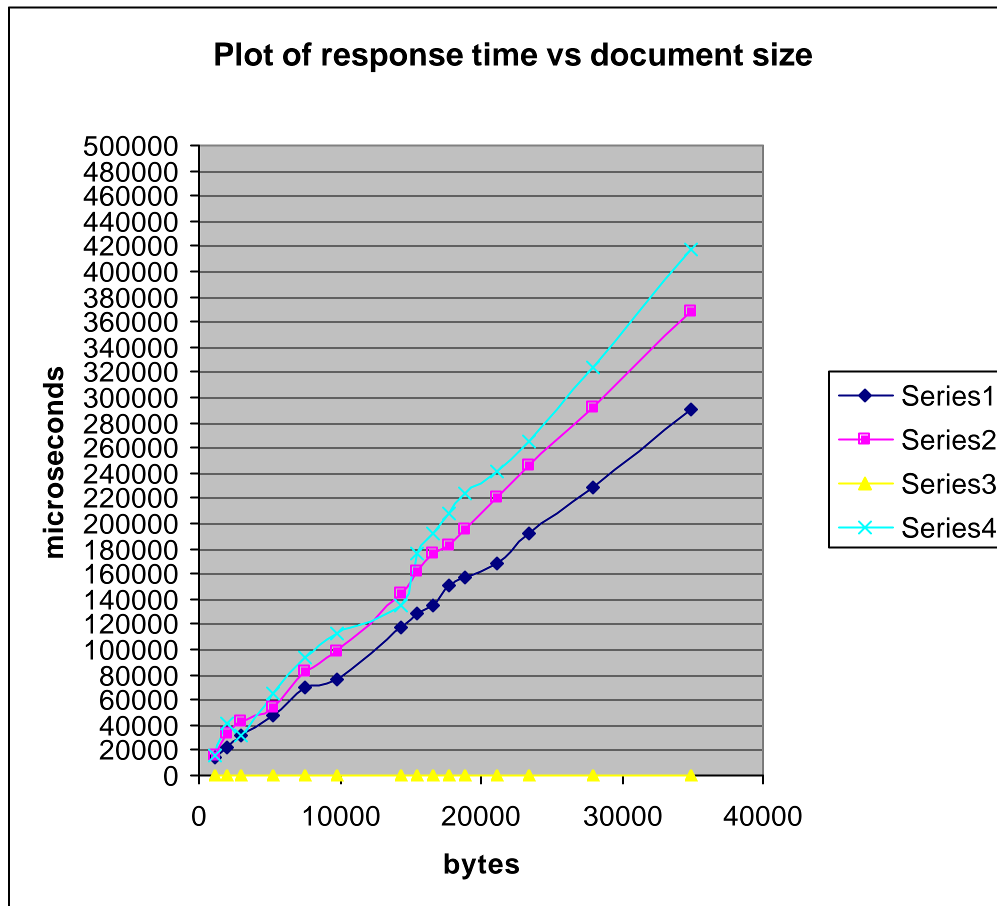


Figure 7. Performance of Pre-allocate Server Scheme

The response time represents the time difference between the time, when the first packet for the request was seen at the content switch and the time, when the first packet of the response from the "correct" real server was seen at the content switch. The document size represents the size of different set of HTTP POST requests used. As shown in the Figure 7 the pre-allocate server scheme with all hits has almost constant response time, whereas with all misses, the response time grows almost exponentially with document size. The comparison between series 1 and series 2 obviously shows the overhead of the rule matching.

# 7 Conclusion

We have presented the design and implementation of a Linux LVS-based content switch called LCS. The impacts of the number of rules, the document size, and the TCP delay binding on the LCS performance are analyzed. We also presented a pre-allocate server scheme to improve the TCP delay binding. Problems encountered during the design of this content switch are discussed and their solutions presented. The performance results of the content switch with the basic TCP delayed binding and that of pre-allocate server scheme are presented. The rule set is implemented as a set of simple if statements with labels and is compiled into a Linux kernel module. The rule module can be

dynamically loaded into the LCS kernel. We also studied the impact of multiple requests of a keep-alive connection on the content switch processing and analyzed as a set of solutions. The software provides a foundation for studying the network and protocol related issues in content switches and cluster systems.

# 8 References

[1] High Performance Cluster Computing: Architectures and Systems, Vol. 1&2, by Rajkumar Buyya (Editor), May 21, 1999, Prentice Hall.
[2] "Linux Virtual Server", http://www.linuxvirtualserver.org.
[3] "Windows 2000 clustering Technologies: Cluster Service Architecture", Microsoft White Paper, 2000. http://www.microsoft.com.
[4] George Apostolopoulos, David Aubespin, Vinod Peris, Prashant Pradhan, Debanjan Saha, " Design, Implementation and Performance of a Content-Based Switch**",** Proc. Infocom2000, Tel Aviv, March 26 - 30, 2000, http://www.ieee-infocom.org/2000/papers/440.ps
[5] Gregory Yerxa and James Hutchinson, "Web Content Switching", http://www.networkcomputing.com.
[6] M. Leech and D. Koblas. SOCKS Protocol Version 5. IETF Request for Comments 1928, April 1996.
[7] D. Maltz and P. Bhagwat. Application Layer Proxy Performance Using TCP Splice. IBM Technical Report RC-21139, March 1998.
[8] "Release Notes for Cisco Content Engine Software". http://www.cisco.com".
[9] "Network-Based Application Recognition Enhancements". http://www.cisco.com.
[10] F5 BIG IP, http://www.f5.com/f5products/bigip/bigipwhitepapers.html.
[11] CISCO Content Services Switch Configuration guide, http://www.cisco.com/univercd/cc/td/doc/product /webscale/css/css_410/advcfggd/index.htm.
[12] "Foundry ServIron Installation and Configuration Guide," May 2000.r
[13] http://www.foundrynetworks.com/techdocs/SI/index.html
[14] "Intel IXA API SDK 4.0 for Intel PA 100," http://www.intel.com/design/network/products/software/ixapi.htm and http://www.intel.com/design/ixa/whitepapers/ixa.htm#IXA_SDK.
[15] Anja Feldmann S. Muthukrishnan "Tradeoffs for Packet Classification**",** Proceedings of Gigabit Networking Workshop GBN 2000, 26 March 2000 - Tel Aviv, Israel **http://www.comsoc.org/socstr/techcom/tcgn/conference/gbn2000/anja-paper.pdf**
[16] Pankaj Gupta and Nick McKcown, "Packet Classification on Multiple Fields", Proc. Sigcomm, September 1999, Harvard University. http://www-cs-students.Stanford.edu/~pankaj/paps/sig99.pdf
[17] V. Srinivasan S. Suri G. Varghese, "Packet Classification using Tuple Space Search", Proc. Sigcomm99, August 30 - September 3, 1999, Cambridge United States, Pages 135 - 146 http://www.acm.org/pubs/articles/proceedings/comm/316188/p135-srinivasan/p135-srinivasan.pdf